Expi2Java Tutorial
Generating Code For The Perrig-Song Protocol

Alex Busenius

July 29, 2009

1 Introduction

This tutorial presents the process of writing a formal specification and generating runnable code for the Perrig-Song mutual authentication protocol[1]. We will start with an informal specification of the protocol and write a formal specification in the Extensible Spi Calculus. We will use an iterative approach, we will write the protocol specification step-by-step and add the needed configurations as we need them. Finally, we will generate Java code implementing the protocol and test the interaction between the participants.

2 Informal Specification

The Perrig-Song protocol is given in Figure 1.

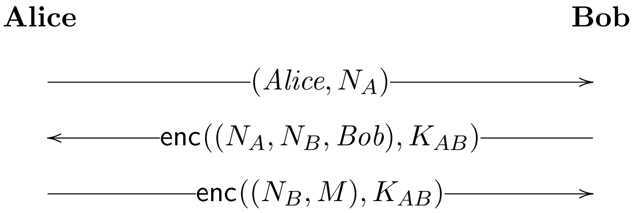
Alice
Bob
$(Alice, N_A)$
$\mathsf{enc}((N_A, N_B, Bob), K_{AB})$
$\mathsf{enc}((N_B, M), K_{AB})$

Figure 1: Perrig-Song mutual authentication protocol

The Perrig-Song protocol uses a shared key to authenticate two participants and send an encrypted message. It is composed of three message exchanges. First, the initiator of the protocol, Alice, sends her identity together with a fresh nonce N_A to the responder, Bob. Bob encrypts the nonce N_A, which he received from Alice together with another fresh nonce N_B and his identity with the shared key K_{AB} and sends this encryption to Alice. Alice receives the encrypted message, decrypts it with the shared key K_{AB} and checks that the nonce N'_A inside is the same as her nonce N_A. If the nonces match, she encrypts the received nonce N_B and a message M

[1] Can be found as one of the examples for CAPSL by J. Millen et al. [MM01], at http://www.csl.sri.com/users/millen/capsl/examples.html

she wanted to send with the shared key K_{AB} and sends the resulting message to Bob. Bob decrypts the message and checks that the received nonce N'_B is the same as his nonce N_B. If the nonces match, the protocol completes successfully.

We will assume that the identities are UTF-8 encoded strings, nonces are randomly chosen 64 bit integers, and the used encryption is AES 256. Further, we will use the usual Java key store mechanism to store the shared key in the file `key.store` that is supposed to be first shared between Alice and Bob in a secure way. For communication we will use TCP/IP, the data will be sent from IP 127.0.0.1 to IP 127.0.0.1 on the port 1234. The localhost is only used here for convenience, the example can be easily extended to use the LAN by specifying the according IP addresses in the configuration. The message that Alice sends to Bob will be an UTF-8 encoded string.

We will generate two programs called `progA` and `progB`, one for each participant. Since TCP/IP assumes one of the participants to be a server and all other to be clients, `progB` will act as a server and `progA` will act as a client. For simplicity, both programs will have a simple command line interface. In a successful protocol run, `progA` will prompt the user to enter the message and `progB` will display the received message. In case of an error the programs will abort with an error message.

3 Formal Specification in Extensible Spi Calculus

We will write the formal specification iteratively, message by message. First, we create two new files in `testData/expi/`, `perrigsong.expi` and `perrigsong.exdef`. To be able to use the provided types and constructors (see the "Input Language" Section of User Manual) we include the file `default.exdef`. We will put all custom configurations into `perrigsong.exdef`, it should also be included. Furthermore, we add two empty processes named pA and pB, and let them be executed in parallel to model the interaction of two participants:

```
(*
 * Perrig-Song mutual authentication protocol
 *
 * http://www.csl.sri.com/users/millen/capsl/examples.html
 * http://sparrow.ece.cmu.edu/~adrian/projects/protgen-csfw/csfw.pdf
 * eXpi version
 *)

include "../exdef/default.exdef"
include "perrigsong.exdef"


let pA =
    0.

let pB =
    0.

process
    (pA | pB)
```

Listing 1: Simple setup: `perrigsong.expi`

The configuration file only contains a comment for now:

```
(*
 * Perrig-Song protocol configuration
 *)
```

Listing 2: Simple setup: `perrigsong.exdef`

3.1 First Message

In the first message Alice sends a concatenation of her identity and a fresh nonce to Bob over a communication channel. To do this, we first need to set up a channel, declare the needed variables and generate a nonce.

In general, constant terms can be modelled in two different ways, as free variables or fresh names. Fresh names are generated using the restriction process **new** a; P and must initialized using configurations. The disadvantage of the restriction processes is however that it can only handle terms of generative types. Free variables are declared using the **free** a construct and can have any type, but must be initialized manually by modifying the generated code.

Identities of both participants are constant strings, we can use the generative type ConstStr and configuration ConstStrCfg to model them, but there is also a specialized configuration IdentifierCfg that is used together with the non-generative type String to model identifiers. As an example, we will use both representations, a free variable alice of type $Identifier as Alice's identity and a fresh name bob of type String as Bob's identity.

In order to declare Alice's identity, we only need to add the declaration line with the type annotation just before **process**:

```
free alice : $Identifier

process
    (pA | pB)
```

Listing 3: Added Alice's identity: `perrigsong.expi`

Generating Bob's identity as a fresh name requires adjusting the ConstStrCfg configuration (we need to set up the value of the generated string). First, we create a configuration called BobIdCfg by extending ConstStrCfg in `perrigsong.exdef`:

```
config BobIdCfg extends ConstStrCfg(
    message = ”bob”
).
```

Listing 4: Added BobIdCfg: `perrigsong.exdef`

Then we create a fresh name bob in the main process (it should be visible in both named processes):

```
process
    new bob : ConstStr@BobIdCfg;
    (pA | pB)
```

Listing 5: Added Bob's identity: `perrigsong.expi`

Now we need to set up the communication channel for the first message. First, we create two TCP/IP channel configurations, one for each participant, and set them up according to the informal specification. We will also need to enter the type of the first message soon, we don't know it at this point, but we can already create a dummy typedef for it.

```
config ChanA extends TcpIpChCfg_( (* client *)
    variable    = "cA",
    host        = "127.0.0.1", (* ip of the server *)
    port        = "1234"
).

config ChanB extends TcpIpChCfg_( (* server *)
    variable    = "cB",
    host        = "any", (* listens on all hosts *)
    port        = "1234"
).

typedef $Msg1 = Top.
```

Listing 6: Channel configurations: `perrigsong.exdef`

Now we create a common channel `c1` and start a server on Bob's side by calling `accept()`. We use the typedef `$Msg1` as the type of the first message.

```
let pB =
    (* listen to connections on c1 *)
    let c1 = accept[->@ChanB](c1) in
    0.
...
process
    new bob : ConstStr@BobIdCfg;
    (* create common communication channel *)
    new c1 : Channel@ChanA<$Msg1>;
    (pA | pB)
```

Listing 7: Created a common channel: `perrigsong.expi`

Finally, we generate a fresh nonce and let Alice send it in the first message. We define a configuration for the 64-bit nonce and a short typedef for convenience:

```
config PSNonceCfg extends NonceCfg(
    size = "8"
).

typedef $PSNonce = Int@PSNonceCfg.
```

Listing 8: Nonce configuration: `perrigsong.exdef`

In Alice's process, we generate a fresh nonce and send the first message. In Bob's process, we receive the message and give both components a name:

```
let pA =
    (* Msg1. A->B: (alice, Na) *)
    new Na : $PSNonce;
    out(c1, (alice, Na)).

let pB =
    ...
    (* Msg1. A->B: (alice, Na) *)
    in(c1, (name, Na)).
```

Listing 9: First message: `perrigsong.expi`

At this point, we can finally correct the type of the first message we preliminary set to Top, it should be ($Identifier , $PSNonce). Additionally, we define a shorter typedef for ConstStr@BobIdCfg and use it in the code. As the result, we get:

```
(*
 * Perrig-Song protocol configuration
 *)

config BobIdCfg extends ASCIIStringCfg(
    message = "bob"
).

config ChanA extends TcpIpChCfg_( (* client *)
    variable    = "cA",
    host        = "127.0.0.1", (* ip of the server *)
    port        = "1234"
).

config ChanB extends TcpIpChCfg_( (* server *)
    variable    = "cB",
    host        = "any", (* listens on all hosts *)
    port        = "1234"
).

config PSNonceCfg extends NonceCfg(
    size = "8"
).


typedef $PSNonce = Int@PSNonceCfg.

typedef $BobId   = ConstStr@BobIdCfg.


typedef $Msg1 = ($Identifier, $PSNonce).
```

Listing 10: Implemented first message: `perrigsong.exdef`

```
(*
 * Perrig-Song mutual authentication protocol
 *
 * http://www.csl.sri.com/users/millen/capsl/examples.html
 * http://sparrow.ece.cmu.edu/~adrian/projects/protgen-csfw/csfw.pdf
 * eXpi version
 *
 * 1. A->B: (alice, Na)
 *)

include "../exdef/default.exdef"
include "perrigsong.exdef"


let pA =
    (* Msg1. A->B: (alice, Na) *)
    new Na : $PSNonce;
    out(c1, (alice, Na)).

let pB =
    (* listen to connections on c1 *)
    let c1 = accept[->@ChanB](c1) in
    (* Msg1. A->B: (alice, Na) *)
    in(c1, (name, Na)).


free alice : $Identifier.

process
    new bob : $BobId;
    (* create common communication channel *)
    new c1 : Channel@ChanA<$Msg1>;
    (pA | pB)
```

Listing 11: Implemented first message: `perrigsong.expi`

filename.expi

Both files can be checked for errors by running expi2java with the --type-check (-t) flag:

```
% ./expi2java -t testData/expi/perrigsong.expi -c -r
parsing...                OK
type checking...          OK
inferring types...        OK
checking configuration... OK
```

The flag --check-config (-c) is needed to also check that all types are configured correctly for code generation.

3.2 Second Message

In the second message, Bob encrypts the received nonce together with a fresh nonce and his identity. Fortunately, the default set of configurations (see the “Configurations” Section of User Manual) already contains the configuration AES that has exactly the settings we need by default. We have already created the configurations for nonce and Bob’s identity, so the only part we still need to care about is the way we get the shared key.

According to the informal specification, we store the key using the Java key store mechanism in the file `key.store`. The default library of cryptographic primitives and data structures provides a special channel configuration KeyStoreChCfg that can be used to retrieve the key. We extend the key store configuration in order to set up the file name. We also define a typedef for the key type, two typedefs for the key store channels and preliminary typedefs for message types (encrypted and plaintext):

```
config PSKeyStoreCfg extends KeyStoreChCfg(
    keystore_filename = "key.store",
).
```

```
typedef $PSKey    = SymKey@AESKeyCfg<Top>.
typedef $PSKStoreRequest = Channel@PSKeyStoreCfg<Top>. (* to send a request *)
typedef $PSKStoreAnswer  = Channel@PSKeyStoreCfg<$PSKey>. (* to receive a key *)

typedef $Msg2Plain  = Top.
typedef $Msg2       = SymEnc@AES<$Msg2Plain>.
```

Listing 12: Key store configuration: `perrigsong.exdef`

We set up the communication channels for the second message in the same way as for the first message:

```
let pB =
    (* listen to connections on c1 *)
    let c1 = accept[->@ChanB](c1) in
    let c2 = accept[->@ChanB](c2) in
    let c3 = accept[->@ChanB](c3) in
    (* Msg1. A->B: (alice, Na) *)
    ...
process
    ...
    (* create common communication channel *)
    new c1 : Channel@ChanA<$Msg1>;
    new c2 : Channel@ChanA<$Msg2>;
    new c3 : Channel@ChanA<$Msg3>;
    (pA | pB)
```

Listing 13: Added remaining channels: `perrigsong.expi`

We send a request for the key to the key store channel, receive a key, generate a fresh nonce and send the second message. We use the received identity to request the key, thus allowing Bob to communicate with other participants, as long as he has the corresponding key:

```
let pB =
    ...
    in(c1, (name, Na));
    (* get the saved key for communication wih Alice *)
    new ks_request : $PSKStoreRequest;
    new ks_answer : $PSKStoreAnswer;
    out(ks_request, name);
    in(ks_answer, Kab);
    (* Msg2. B->A: enc((Na, Nb, bob), K_AB) *)
    new Nb : $PSNonce;
    out(c2, enc((Na, Nb, bob), Kab)).
```

Listing 14: Sent second message: `perrigsong.expi`

Now we can change the $Msg2Plain typedef to the correct type:

```
typedef $Msg2Plain  = ($PSNonce, $PSNonce, $BobId).
```

Listing 15: Key store configuration: `perrigsong.exdef`

In Alice's process we first get the stored shared key for the communication with Bob in the same way as before, then receive the message, decrypt it and split the pairs:

```
let pA =
    ...
    out(c1, (alice, Na));
    (* get the saved key for communication with Bob *)
    new ks_request : $PSKStoreRequest;
    new ks_answer : $PSKStoreAnswer;
    out(ks_request, bob);
    in(ks_answer, Kab);
    (* Msg2. B->A: enc((Na, Nb, bob), K_AB) *)
    in(c2, msg2);
    let (Nx, Nb, name) = dec(msg2, Kab) in
        0.
```

Listing 16: Key store configuration: `perrigsong.exdef`

Now Alice must check that the received name is "Bob" and the first nonce equals the one she sent to Bob in the first message, otherwise we abort the protocol:

```
let pA =
    ...
    let (Nx, Nb, name) = dec(msg2, Kab) in
        (* check received data *)
        let ok = eq((name, Nx), (bob, Na)) in
            0.
```

Listing 17: Checked received data: `perrigsong.expi`

The result after implementing two messages is as follows:

```
(*
 * Perrig-Song protocol configuration
 *)

config BobIdCfg extends ASCIIStringCfg(
    message = "bob"
).

config ChanA extends TcpIpChCfg_( (* client *)
    variable    = "cA",
    host        = "127.0.0.1", (* ip of the server *)
    port        = "1234"
).

config ChanB extends TcpIpChCfg_( (* server *)
    variable    = "cB",
    host        = "any", (* listens on all hosts *)
    port        = "1234"
).

config PSNonceCfg extends NonceCfg(
    size = "8"
).

config PSKeyStoreCfg extends KeyStoreChCfg(
    keystore_filename = "key.store"
).


```

```
typedef $PSNonce = Int@PSNonceCfg.

typedef $BobId   = ConstStr@BobIdCfg.

typedef $PSKey   = SymKey@AES<Top>.

typedef $PSKStoreRequest = Channel@PSKeyStoreCfg<Top>. (* to send a request *)
typedef $PSKStoreAnswer  = Channel@PSKeyStoreCfg<$PSKey>. (* to receive a key *)


typedef $Msg1      = ($Identifier, $PSNonce).
typedef $Msg2Plain = ($PSNonce, $PSNonce, $BobId).
typedef $Msg2      = SymEnc@AES<$Msg2Plain>.
typedef $Msg3Plain = Top.
typedef $Msg3      = SymEnc@AES<$Msg3Plain>.
```

Listing 18: Implemented second message: `perrigsong.exdef`

```
(*
 * Perrig-Song mutual authentication protocol
 *
 * http://www.csl.sri.com/users/millen/capsl/examples.html
 * http://sparrow.ece.cmu.edu/~adrian/projects/protgen-csfw/csfw.pdf
 * eXpi version
 *
 * 1. A->B: (alice, Na)
 * 2. B->A: enc((Na, Nb, bob), Kab)
 *)

include "../exdef/default.exdef"
include "perrigsong.exdef"


let pA =
    (* Msg1. A->B: (alice, Na) *)
    new Na : $PSNonce;
    out(c1, (alice, Na));
    (* get the saved key for communication wih Bob *)
    new ks_request : $PSKStoreRequest;
    new ks_answer : $PSKStoreAnswer;
    out(ks_request, bob);
    in(ks_answer, Kab);
    (* Msg2. B->A: enc((Na, Nb, bob), K_AB) *)
    in(c2, msg2);
    let (Nx, Nb, name) = dec(msg2, Kab) in
        (* check received data *)
        let ok = eq((name, Nx), (bob, Na)) in
            0.

let pB =
    (* listen to connections on c1 *)
    let c1 = accept[->@ChanB](c1) in
    let c2 = accept[->@ChanB](c2) in
    let c3 = accept[->@ChanB](c3) in
    (* Msg1. A->B: (alice, Na) *)
    in(c1, (name, Na));
    (* get the saved key for communication wih Alice *)
    new ks_request : $PSKStoreRequest;
    new ks_answer : $PSKStoreAnswer;
    out(ks_request, name);
    in(ks_answer, Kab);
    (* Msg2. B->A: enc((Na, Nb, bob), K_AB) *)
    new Nb : $PSNonce;
    out(c2, enc((Na, Nb, bob), Kab));
    0.


free alice : $Identifier.

process
    new bob : $BobId;
    (* create common communication channel *)
    new c1 : Channel@ChanA<$Msg1>;
    new c2 : Channel@ChanA<$Msg2>;
    new c3 : Channel@ChanA<$Msg3>;
    (pA | pB)
```

Listing 19: Implemented second message: `perrigsong.expi`

3.3 Third Message

In third message, Alice encrypts the previously received nonce Nb together with the text message and sends them to Bob. We first need to prompt the user to enter the message and assign the entered text to a variable that we can then encrypt. In order to access the console, we use the channel configuration ConsoleChCfg and the corresponding typedefs $STDIN and $STDOUT. For the message type we will use String, since it is UTF-8 encoded by default.

We extend the ConstStrCfg to define the prompt message and define the type that the third message is going to have:

```
config UserPrompt extends ConstStrCfg(message = "Please␣enter␣the␣message:␣").
typedef $PromptMsg  = ConstStr@UserPrompt.

typedef $Msg3Plain  = ($PSNonce, String).
```

Listing 20: Console configuration: `perrigsong.exdef`

In Alice's process, we set up the console channels, generate the prompt message, show it to the user and enter the message. Afterwards we complete Alice's process by sending the third message:

```
let pA =
    ...
    let ok = eq(Nx, Na) in
        (* User interaction *)
        new cin : $STDIN;
        new cout : $STDOUT;
        new prompt : $PromptMsg;
        out(cout, prompt);
        in(cin, message);
        (* Msg3. A->B: enc((Nb, M), Kab) *)
        out(c3, enc((Nb, message), Kab)).
...
process
```

```
    new bob : $BobId;
    (* console channels *)
    new cin : $STDIN;
    new cout : $STDOUT;
    ...
```

Listing 21: Sent the third message: `perrigsong.expi`

In Bob's process, we receive the message, decrypt it, split the resulting pair, check that the received nonce equals the one just sent and display the message to the user:

```
let pB =
    ...
    (* Msg3. A->B: enc((Nb, M), Kab) *)
    in(c3, msg3);
    let (Nx, message) = dec(msg3, Kab) in
        let ok = eq(Nx, Nb) in
            (* show the received message *)
            out(cout, message).
```

Listing 22: Received the third message: `perrigsong.expi`

Finally, we get the complete implementation of the Perrig-Song mutual authentication protocol in Extensible Spi Calculus:

```
(*
 * Perrig-Song protocol configuration
 *)

config BobIdCfg extends ASCIIStringCfg(
    message = "bob"
).

config ChanA extends TcpIpChCfg_( (* client *)
    variable     = "cA",
    host         = "127.0.0.1", (* ip of the server *)
    port         = "1234"
).

config ChanB extends TcpIpChCfg_( (* server *)
    variable     = "cB",
    host         = "any", (* listens on all hosts *)
    port         = "1234"
).

config PSNonceCfg extends NonceCfg(
    size = "8"
).

config PSKeyStoreCfg extends KeyStoreChCfg(
    keystore_filename = "key.store"
).


typedef $PSNonce = Int@PSNonceCfg.

typedef $BobId   = ConstStr@BobIdCfg.

typedef $PSKey    = SymKey@AES<Top>.

typedef $PSKStoreRequest = Channel@PSKeyStoreCfg<Top>. (* to send a request *)
typedef $PSKStoreAnswer  = Channel@PSKeyStoreCfg<$PSKey>. (* to receive a key *)

config UserPrompt extends ConstStrCfg(message = "Please␣enter␣the␣message:␣").
typedef $PromptMsg  = ConstStr@UserPrompt.

typedef $Msg1       = ($Identifier, $PSNonce).
typedef $Msg2Plain  = ($PSNonce, $PSNonce, $BobId).
typedef $Msg2       = SymEnc@AES<$Msg2Plain>.
typedef $Msg3Plain  = ($PSNonce, String).
typedef $Msg3       = SymEnc@AES<$Msg3Plain>.
```

Listing 23: Perrig-Song mutual authentication protocol: `perrigsong.exdef`

```
(*
 * Perrig-Song mutual authentication protocol
 *
 * http://www.csl.sri.com/users/millen/capsl/examples.html
 * http://sparrow.ece.cmu.edu/~adrian/projects/protgen-csfw/csfw.pdf
 * eXpi version
 *
 * 1. A->B: (alice, Na)
 * 2. B->A: enc((Na, Nb, bob), Kab)
 * 3. A->B: enc((Nb, message), Kab)
 *)

include "../exdef/default.exdef"
include "perrigsong.exdef"


let pA =
    (* Msg1. A->B: (alice, Na) *)
    new Na : $PSNonce;
    out(c1, (alice, Na));
    (* get the saved key for communication wih Bob *)
    new ks_request : $PSKStoreRequest;
    new ks_answer : $PSKStoreAnswer;
    out(ks_request, bob);
    in(ks_answer, Kab);
    (* Msg2. B->A: enc((Na, Nb, bob), K_AB) *)
    in(c2, msg2);
    let (Nx, Nb, name) = dec(msg2, Kab) in
        (* check received data *)
        let ok = eq((name, Nx), (bob, Na)) in
            (* User interaction *)
            new prompt : $PromptMsg;
            out(cout, prompt);
            in(cin, message);
            (* Msg3. A->B: enc((Nb, M), Kab) *)
            out(c3, enc((Nb, message), Kab)).

let pB =
    (* listen to connections on c1 *)
    let c1 = accept[->@ChanB](c1) in
    let c2 = accept[->@ChanB](c2) in
    let c3 = accept[->@ChanB](c3) in
    (* Msg1. A->B: (alice, Na) *)
    in(c1, (name, Na));
    (* get the saved key for communication wih Alice *)
    new ks_request : $PSKStoreRequest;
    new ks_answer : $PSKStoreAnswer;
    out(ks_request, name);
    in(ks_answer, Kab);
    (* Msg2. B->A: enc((Na, Nb, bob), K_AB) *)
    new Nb : $PSNonce;
    out(c2, enc((Na, Nb, bob), Kab));
    (* Msg3. A->B: enc((Nb, M), Kab) *)
    in(c3, msg3);
    let (Nx, message) = dec(msg3, Kab) in
        let ok = eq(Nx, Nb) in
            (* show the received message *)
            out(cout, message).


free alice : $Identifier.

process
    new bob : $BobId;
    (* console channels *)
    new cin : $STDIN;
    new cout : $STDOUT;
    (* create common communication channel *)
    new c1 : Channel@ChanA<$Msg1>;
    new c2 : Channel@ChanA<$Msg2>;
    new c3 : Channel@ChanA<$Msg3>;
    (pA | pB)
```

Listing 24: Perrig-Song mutual authentication protocol: `perrigsong.expi`

4 Code Generation

The code generation step is automated, only some configuration options must be specified by the user. We will use the `testCode/expi2java/perrigsong` directory as the target directory, and “perrigsong” as the package name. The implementation class will be called “PerrigSong”. We will use the default code templates in `data/expi2java/javaTemplates` and no additional imports, since we do not use any additional implementation classes. In addition, the output directory must be created first. This results in the following calls:

```
% mkdir testCode/expi2java/perrigsong
% ./expi2java -g testData/expi/perrigsong.expi \
              -p perrigsong \
              -m data/expi2java/javaTemplates \
              -o testCode/expi2java/perrigsong \
              -n "PerrigSong"
parsing...                 OK
type checking...           OK
inferring types...         OK
checking configuration...  OK
generating code...         OK
```

Since we decided to use a free name for the Alice's identity in Section 3.1, we need to initialize all free names in the generated code.

We go to the code directory and open the file `perrigsong/Application.java` and search for the word "FIXME". In the lines 102-103 we find the following code:

```
    // FIXME all free names are user-provided, initialize and finalize them ...
    Identifier alice_139 = null;
```

Listing 25: Generated `perrigsong/Application.java`

We change it to:

```
    Identifier alice_139 = new Identifier("alice");
```

Listing 26: Initialized free names in `perrigsong/Application.java`

Now we need to compile the generated code. We set the class path to contain the library and the Bouncy Castle Provider Jar file and run the compiler:

```
% cd testCode/expi2java
% CLASSPATH=".:../../bin/eXpiLibrary.jar:../../lib/bcprov-jdk16-141.jar"
% javac perrigsong/*.java
```

Now we can start the generated test application:

```
% java perrigsong.Application --help

eXpi2Java protocol test - Run the generated PerrigSong protocol

USAGE: -h [options]
  OPTIONS:
    -h, --help                  : Print this help
    -d, --debug                 : Enable debug modus
    --show                      : Show the content of the keystore (buggy)
    -k <name> <config>
     --keygen <name> <config>   : Generate a new symmetric key, store it under
                                  this name and exit
    -c <from> <to> <config>
     --copy-key <from> <to> <config>
                                : Copy a symmetric key stored under the name
                                  <from> as <to> and exit
    -p <name>
     --participant <name>       : Only run code for this participant
                                  <name> can be on of: pA, pB
```

The default test application offer several options. If started without parameters, it runs both participants in parallel, to run only one of the participants, the option --participant can be used. With --debug, some additional information about the protocol progress is shown. With --keygen and --copy-key, we can modify the contents of a key store.

Since the Perrig-Song protocol relies on the known shared key, we first need to create it and store it in the key store:

```
% java perrigsong.Application -k alice AESKeyCfg
% java perrigsong.Application -c alice bob AESKeyCfg
% ls jce*
jce.keystore
% mv jce.keystore key.store
% ls key*
key.store
```

Note that `jce.keystore` is used as the key store filename by default and we need to rename it first to match our settings.

Now we can finally test the generated protocol. We open another shell, go to the code directory, set up the class path as before and start the server:

```
% cd testCode/expi2java
% CLASSPATH=".:../../bin/eXpiLibrary.jar:../../lib/bcprov-jdk16-141.jar"
% java perrigsong.Application -p pB
```

In the previous shell, we start the client, after some short time, a prompt should appear. We enter the message “’Hello!!!’ and press enter:

```
% java perrigsong.Application -p pA
Please enter the message:
Hello!!!
```

This message should be printed on the shell where the server is running, then both programs should terminate:

```
% java perrigsong.Application -p pB
Hello!!!
```

References

[MM01] J. Millen and F. Muller. Cryptographic protocol generation from CAPSL. Technical Report SRI-CSL-01-07, SRI International, December 2001. 1

List of Figures

Listings